Dynamic Trees

- Goal: maintain a forest of rooted trees with costs on vertices.
  - Each tree has a root, every edge directed towards the root.

- Operations allowed:
  - \textbf{link}(v,w): creates an edge between \( v \) (a root) and \( w \).
  - \textbf{cut}(v,w): deletes edge \((v,w)\).
  - \textbf{findcost}(v): returns the cost of vertex \( v \).
  - \textbf{findroot}(v): returns the root of the tree containing \( v \).
  - \textbf{findmin}(v): returns the vertex \( w \) of minimum cost on the path from \( v \) to the root (if there is a tie, choose the closest to the root).
  - \textbf{addcost}(v,x): adds \( x \) to the cost every vertex from \( v \) to root.
Dynamic Trees

- An example (two trees):
Dynamic Trees

link(q,e)

Dynamic Trees
Dynamic Trees

\[ \text{cut}(q) \]
Dynamic Trees

- $\text{findmin}(s) = b$
- $\text{findroot}(s) = a$
- $\text{findcost}(s) = 2$
- $\text{addcost}(s, 3)$
Obvious Implementation

- A node represents each vertex;
- Each node \( x \) points to its parent \( p(x) \):
  - cut, split, findcost: constant time.
  - findroot, findmin, addcost: linear time on the size of the path.
- Acceptable if paths are small, but \( O(n) \) in the worst case.
- Cleverer data structures achieve \( O(\log n) \) for all operations.
Simple Paths

- We start with a simpler problem:
  - Maintain **set of paths** subject to:
    - **split**: cuts a path in two;
    - **concatenate**: links endpoints of two paths, creating a new path.
  - Operations allowed:
    - **findcost**(v): returns the cost of vertex v;
    - **addcost**(v, x): adds x to the cost of vertices in path containing v;
    - **findmin**(v): returns minimum-cost vertex path containing v.
Simple Paths as Lists

- Natural representation: **doubly linked list**.
  - Constant time for **findcost**.
  - Constant time for **concatenate** and **split** if endpoints given, linear time otherwise.
  - Linear time for **findmin** and **addcost**.

- Can we do it \(O(\log n)\) time?

---

costs: 6 2 3 4 7 9 3
\(v_1\) \(v_2\) \(v_3\) \(v_4\) \(v_5\) \(v_6\) \(v_7\)
Simple Paths as Binary Trees

- Alternative representation: **balanced binary trees**.
  - **Leaves**: vertices in symmetric order.
  - **Internal nodes**: subpaths between extreme descendants.
Simple Paths as Binary Trees

- **Compact alternative:**
  - Each *internal node* represents both a *vertex* and a *subpath*:
    - subpath from leftmost to rightmost descendant.
Simple Paths: Maintaining Costs

• Keeping costs:
  - First idea: store $\text{cost}(x)$ directly on each vertex;
  - Problem: addcost takes linear time (must update all vertices).

```
actual costs
9   v_6
/  \
2   v_2
/  \
6   v_1
/     \
4   v_3
/   \
3   v_4
/     \
7   v_5
```

```
costs: 6   2   3   4   7   9   3
v_1  v_2  v_3  v_4  v_5  v_6  v_7
```
Better approach: store $\Delta cost(x)$ instead:
- Root: $\Delta cost(x) = cost(x)$
- Other nodes: $\Delta cost(x) = cost(x) - cost(p(x))$
Simple Paths: Maintaining Costs

- Costs:
  - `addcost`: constant time (just add to root)
  - Finding `cost(x)` is slightly harder: $O(\text{depth}(x))$.

![Diagram of actual and difference form costs](image)
Simple Paths: Finding Minima

- Still have to implement \texttt{findmin}:
  - Store $\text{mincost}(x)$, the minimum cost on subpath with root $x$.
  - \texttt{findmin} runs in $O(\log n)$ time, but \texttt{addcost} is linear.

- Actual costs:

- Costs:

- Dynamic Trees
Simple Paths: Finding Minima

- Store $\Delta \min(x) = \text{cost}(x) - \text{mincost}(x)$ instead.
  - `findmin` still runs in $O(\log n)$ time, `addcost` now constant.
Simple Paths: Data Fields

- Final version:
  - Stores $\Delta \text{min}(x)$ and $\Delta \text{cost}(x)$ for every vertex

```latex
\begin{itemize}
  \item \text{actual costs:}\n    \begin{itemize}
      \item $v_1$: 6
      \item $v_2$: 2
      \item $v_3$: 3
      \item $v_4$: 4
      \item $v_5$: 7
      \item $v_6$: 9
    \end{itemize}
  \item \text{(\Delta \text{cost}, \Delta \text{min})}\n    \begin{itemize}
      \item $v_1$: (4,0)
      \item $v_2$: (-7,0)
      \item $v_3$: (1,0)
      \item $v_4$: (2,0)
      \item $v_5$: (3,0)
      \item $v_6$: (9, 7)
      \item $v_7$: (-6,0)
    \end{itemize}
\end{itemize}
```
Simple Paths: Structural Changes

- **Concatenating** and **splitting** paths:
  - Join or split the corresponding binary trees;
  - Time proportional to tree **height**.
  - For **balanced** trees, this is $O(\log n)$.
    - **Rotations** must be supported in constant time.
    - We must be able to update $\Delta\text{min}$ and $\Delta\text{cost}$.
Simple Paths: Structural Changes

• Restructuring primitive: \textit{rotation}.

\[\begin{align*}
\Delta \text{cost}'(v) &= \Delta \text{cost}(v) + \Delta \text{cost}(w) \\
\Delta \text{cost}'(w) &= -\Delta \text{cost}(v) \\
\Delta \text{cost}'(b) &= \Delta \text{cost}(v) + \Delta \text{cost}(b) \\
\Delta \text{min}'(w) &= \max\{0, \Delta \text{min}(b) - \Delta \text{cost}'(b), \Delta \text{min}(c) - \Delta \text{cost}(c)\} \\
\Delta \text{min}'(v) &= \max\{0, \Delta \text{min}(a) - \Delta \text{cost}(a), \Delta \text{min}'(w) - \Delta \text{cost}'(w)\}
\end{align*}\]
Splaying

- Simpler alternative to balanced binary trees: **splaying**.
  - Does not guarantee that trees are balanced in the worst case.
  - Guarantees $O(\log n)$ access in the **amortized** sense.
  - Makes the data structure much **simpler** to implement.

- Basic characteristics:
  - Does not require any balancing information;
  - On an access to $v$, **splay** on $v$:
    - Moves $v$ to the **root**;
    - Roughly **halves** the depth of other nodes in the access path.
  - Based entirely on **rotations**.

- Other operations (**insert**, **delete**, **join**, **split**) use splay.
Splaying

- Three restructuring operations:

  1. **zigzag(x)**
  2. **zigzig(x)**
  3. **zig(x)**

(only happens if y is root)
An Example of Splaying
An Example of Splaying

Dynamic Trees
An Example of Splaying

Dynamic Trees
An Example of Splaying

Dynamic Trees
An Example of Splaying
An Example of Splaying

Dynamic Trees
An Example of Splaying

Dynamic Trees
An Example of Splaying

Dynamic Trees
An Example of Splaying
An Example of Splaying

- End result:

```
Dynamic Trees
```

```
An Example of Splaying
```

```
End result:
```

```
Dynamic Trees
```
Amortized Analysis

- Bounds the running time of a sequence of operations.
- **Potential function** $\Phi$ maps each configuration to real number.
- **Amortized time** to execute each operation:
  - $a_i = t_i + \Phi_i - \Phi_{i-1}$
    - $a_i$: amortized time to execute $i$-th operation;
    - $t_i$: actual time to execute the operation;
    - $\Phi_i$: potential after the $i$-th operation.
- Total time for $m$ operations:
  \[
  \sum_{i=1..m} t_i = \sum_{i=1..m} (a_i + \Phi_{i-1} - \Phi_i) = \Phi_0 - \Phi_m + \sum_{i=1..m} a_i
  \]
Amortized Analysis of Splaying

• Definitions:
  ▪ \( s(x) \): size of node \( x \) (number of descendants, including \( x \));
    • At most \( n \), by definition.
  ▪ \( r(x) \): rank of node \( x \), defined as \( \log s(x) \);
    • At most \( \log n \), by definition.
  ▪ \( \Phi_i \): potential of the data structure (\text{twice} the sum of all ranks).
    • At most \( O(n \log n) \), by definition.

• Access Lemma [ST85]: The amortized time to splay a tree with root \( t \) at a node \( x \) is at most

\[
6(r(t) - r(x)) + 1 = O(\log(s(t)/s(x))).
\]
Proof of Access Lemma

- **Access Lemma** [ST85]: *The amortized time to splay a tree with root $t$ at a node $x$ is at most*

  \[ 6(r(t) - r(x)) + 1 = O(\log(s(t)/s(x))). \]

- **Proof idea:**
  - $r_i(x) = \text{rank of } x \text{ after the } i\text{-th splay step}$;
  - $a_i = \text{amortized cost of the } i\text{-th splay step}$;
  - $a_i \leq 6(r_i(x) - r_{i-1}(x)) + 1$ (for the zig step, if any)
  - $a_i \leq 6(r_i(x) - r_{i-1}(x))$ (for any zig-zig and zig-zag steps)
  - Total amortized time for all $k$ steps:
    \[
    \sum_{i=1..k} a_i \leq \sum_{i=1..k-1} [6(r_i(x) - r_{i-1}(x))] + [6(r_k(x) - r_{k-1}(x)) + 1] \\
    = 6r_k(x) - 6r_o(x) + 1
    \]
Proof of Access Lemma: Splaying Step

• Zig-zig:

Claim: \( a \leq 6 \left( r'(x) - r(x) \right) \)

\[ t + \Phi' - \Phi \leq 6 \left( r'(x) - r(x) \right) \]

\[ 2 + 2(r'(x) + r'(y) + r'(z)) - 2(r(x) + r(y) + r(z)) \leq 6 \left( r'(x) - r(x) \right) \]

\[ 1 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \leq 3 \left( r'(x) - r(x) \right) \]

\[ 1 + r'(y) + r'(z) - r(x) - r(y) \leq 3 \left( r'(x) - r(x) \right) \quad \text{since } r'(x) = r(z) \]

\[ 1 + r'(y) + r'(z) - 2r(x) \leq 3 \left( r'(x) - r(x) \right) \quad \text{since } r(y) \geq r(x) \]

\[ 1 + r'(x) + r'(z) - 2r(x) \leq 3 \left( r'(x) - r(x) \right) \quad \text{since } r'(x) \geq r'(y) \]

\[ (r(x) - r'(x)) + (r'(z) - r'(x)) \leq -1 \quad \text{rearranging} \]

\[ \log(s(x)/s'(x)) + \log(s'(z)/s'(x)) \leq -1 \quad \text{definition of rank} \]

TRUE because \( s(x) + s'(z) < s'(x) \): both ratios are smaller than 1, at least one is at most 1/2.
Proof of Access Lemma: Splaying Step

- Zig-zag:

Claim: $a \leq 4 \left( r'(x) - r(x) \right)$

$t + \Phi' - \Phi \leq 4 \left( r'(x) - r(x) \right)$

$2 + (2r'(x) + 2r'(y) + 2r'(z)) - (2r(x) + 2r(y) + 2r(z)) \leq 4 \left( r'(x) - r(x) \right)$

$2 + 2r'(y) + 2r'(z) - 2r(x) - 2r(y) \leq 4 \left( r'(x) - r(x) \right),$ \hspace{1em} since $r'(x) = r(z)$

$2 + 2r'(y) + 2r'(z) - 4r(x) \leq 4 \left( r'(x) - r(x) \right),$ \hspace{1em} since $r(y) \geq r(x)$

$(r'(y) - r'(x)) + (r'(z) - r'(x)) \leq -1,$ \hspace{1em} rearranging

$log(s'(y)/s'(x)) + log(s'(z)/s'(x)) \leq -1$ \hspace{1em} definition of rank

TRUE because $s'(y) + s'(z) < s'(x)$: both ratios are smaller than 1, at least one is at most 1/2.
Proof of Access Lemma: Splaying Step

- **Zig:**

Claim: \( a \leq 1 + 6 \,(r'(x) - r(x)) \)

\( t + \Phi' - \Phi \leq 1 + 6 \,(r'(x) - r(x)) \)

\( 1 + (2r'(x) + 2r'(y)) - (2r(x) + 2r(y)) \leq 1 + 6 \,(r'(x) - r(x)) \)

\( 1 + 2 \,(r'(x) - r(x)) \leq 1 + 6 \,(r'(x) - r(x)) \), since \( r(y) \geq r'(y) \)

TRUE because \( r'(x) \geq r(x) \).
Splaying

• To sum up:
  - No rotation: $a = 1$
  - Zig: $a \leq 6 \left( r'(x) - r(x) \right) + 1$
  - Zig-zig: $a \leq 6 \left( r'(x) - r(x) \right)$
  - Zig-zag: $a \leq 4 \left( r'(x) - r(x) \right)$
  - Total amortized time at most $6 \left( r(t) - r(x) \right) + 1 = \mathcal{O}(\log n)$

• Since accesses bring the relevant element to the root, other operations (insert, delete, join, split) become trivial.
Dynamic Trees

- We know how to deal with isolated paths.
- How to deal with paths within a tree?
Dynamic Trees

- Main idea: partition the vertices in a tree into disjoint **solid** paths connected by **dashed** edges.
Dynamic Trees

- Main idea: partition the vertices in a tree into disjoint **solid paths** connected by **dashed** edges.
Dynamic Trees

- A vertex $v$ is exposed if:
  - There is a solid path from $v$ to the root;
  - No solid edge enters $v$. 
Dynamic Trees

- A vertex $v$ is **exposed** if:
  - There is a *solid path* from $v$ to the *root*;
  - *No solid edge* enters $v$.

- It is **unique**.
Dynamic Trees

• **Solid paths:**
  - Represented as *binary trees* (as seen before).
  - *Parent pointer of root* is the outgoing dashed edge.
  - Hierarchy of solid binary trees linked by dashed edges: "*virtual tree*".

• "Isolated path" operations handle the *exposed* path.
  - The solid path entering the root.
  - Dashed pointers go *up*, so the solid path does not "know" it has dashed children.

• If a different path is needed:
  - *expose*(\(v\)): make entire path from \(v\) to the root solid.
Virtual Tree: An Example

Actual tree

Virtual tree
Dynamic Trees

- Example: $\text{expose}(v)$
Example: $\text{expose}(v)$

- Take all edges in the path to the root, ...
Dynamic Trees

- Example: \texttt{expose(\textit{v})}
  - ..., make them solid, ...
Dynamic Trees

- Example: `expose(ν)`
  - ...make sure there is no other solid edge incident into the path.
  - Uses `splice` operation.
Exposing a Vertex

- **expose**(x): makes the path from x to the root solid.

- Implemented in three steps:
  1. Splay within each solid tree in the path from x to root.
  2. Splice each dashed edge from x to the root.
     - **splice** makes a dashed become the left solid child;
     - If there is an original left solid child, it becomes dashed.
  3. Splay on x, which will become the root.
Exposing a Vertex: An Example

• $\text{expose}(a)$

(virtual trees)
Dynamic Trees: Splice

- Additional restructuring primitive: *splice*.

- Updates:
  - $\Delta cost'(v) = \Delta cost(v) - \Delta cost(z)$
  - $\Delta cost'(u) = \Delta cost(u) + \Delta cost(z)$
  - $\Delta min'(z) = \max\{0, \Delta min(v) - \Delta cost'(v), \Delta min(x) - \Delta cost(x)\}$

- Will only occur when $z$ is the root of a tree.
Exposing a Vertex: Running Time

- Running time of \texttt{expose}(x):
  - proportional to initial depth of \( x \);
  - \( x \) is rotated all the way to the root;
  - we just need to count the number of rotations;
    - will actually find amortized number of rotations: \( \mathcal{O}(\log n) \).
  - proof uses the Access Lemma.
    - \( s(x) \), \( r(x) \) and potential are defined as before;
    - In particular, \( s(x) \) is the size of the whole subtree rooted at \( x \);
      - Includes both solid and dashed edges.
Exposing a Vertex: Running Time (Proof)

- **k**: number of dashed edges from \( x \) to the root \( t \).

- **Amortized costs of each pass:**
  1. Splay within each solid tree:
     - \( x_i \): vertex splayed on the \( i \)-th solid tree.
     - Amortized cost of \( i \)-th splay: \( 6 (r'(x_i) - r(x_i)) + 1 \).
     - \( r(x_{i+1}) \geq r'(x_i) \), so the sum over all steps telescopes;
     - Amortized cost first of pass: \( 6(r'(x_k) - r(x_1)) + k \leq 6 \log n + k \).
  2. Splice dashed edges:
     - no rotations, no potential changes: amortized cost is zero.
  3. Splay on \( x \):
     - Amortized cost is at most \( 6 \log n + 1 \).
     - \( x \) ends up in root, so exactly \( k \) rotations happen;
     - each rotation costs one credit, but is charged two;
     - they pay for the extra \( k \) rotations in the first pass.

- Amortized number of rotations = \( O(\log n) \).
Implementing Dynamic Tree Operations

- **findcost(\(v\))**:
  - expose \(v\), return \(cost(\(v\)).\)

- **findroot(\(v\))**:
  - expose \(v\);
  - find \(w\), the rightmost vertex in the solid subtree containing \(v\);
  - splay at \(w\) and return \(w\).

- **findmin(\(v\))**:
  - expose \(v\);
  - use \(\Delta cost\) and \(\Delta min\) to walk down from \(v\) to \(w\), the last minimum-cost node in the solid subtree;
  - splay at \(w\) and return \(w\).
Implementing Dynamic Tree Operations

- **addcost**(v, x):
  - expose v;
  - add x to Δcost(v);

- **link**(v, w):
  - expose v and w (they are in different trees);
  - set p(v) = w (that is, make v a middle child of w).

- **cut**(v):
  - expose v;
  - add Δcost(v) to Δcost(right(v));
  - make p(right(v)) = null and right(v) = null.
Extensions and Variants

- Simple extensions:
  - Associate values with edges:
    - just interpret $\text{cost}(v)$ as $\text{cost}(v,p(v))$.
  - other path queries (such as length):
    - change values stored in each node and update operations.
  - free (unrooted) trees.
    - implement $\text{evert}$ operation, which changes the root.

- Not-so-simple extension:
  - subtree-related operations:
    - requires that vertices have bounded degree;
    - Approach for arbitrary trees: “ternarize” them:
      - [Goldberg, Grigoriadis and Tarjan, 1991]
Alternative Implementation

- Total time per operation depends on the data structure used to represent paths:
  - Splay trees: $O(\log n)$ amortized [ST85].
  - Balanced search tree: $O(\log^2 n)$ amortized [ST83].
  - Locally biased search tree: $O(\log n)$ amortized [ST83].
  - Globally biased search trees: $O(\log n)$ worst-case [ST83].

- Biased search trees:
  - Support leaves with different “weights”.
  - Some solid leaves are “heavier” because they also represent subtrees dangling from it from dashed edges.
  - Much more complicated than splay trees.
Other Data Structures

- Some applications require tree-related information:
  - minimum vertex in a tree;
  - add value to all elements in the tree;
  - link and cut as usual.

- **ET-Trees** can do that:
  - Henzinger and King (1995);
  - Tarjan (1997).
ET-Trees

- Each tree represented by its *Euler tour*.
  - Edge \( \{v,w\} \):
    - appears as arcs \((v,w)\) and \((w,v)\)
  - Vertex \(v\):
    - appears once as a self-loop \((v,v)\):
    - used as an “anchor” for new links.
    - stores vertex-related information.
  - Representation is not circular: tour broken at arbitrary place.
ET-Trees

- Consider $\text{link}(v,w)$:
  - Create elements representing arcs $(v,w)$ and $(w,v)$:
    
    $$(v,w) \quad (w,v)$$

  - Split and concatenate tours appropriately:
    - Original tours:
      
      \[
      L_v \xrightarrow{v} L'_v \quad \text{and} \quad L_w \xrightarrow{w} L'_w
      \]
    - Final tour:
      
      \[
      L_v \xrightarrow{v} (v,w) \xrightarrow{L'_w} L_w \xrightarrow{w} (w,v) \xrightarrow{L'_v}
      \]

- The cut operation is similar.
ET-Trees

- Tours as doubly-linked lists:
  - Natural representation.
  - \textbf{link/cut}: $O(1)$ time.
  - \textbf{addcost/findmin}: $O(n)$ time.

- Tours as balanced binary search trees:
  - \textbf{link/cut}: $O(\log n)$ time (binary tree join and split).
  - \textbf{addcost/findmin}: $O(\log n)$ time:
    - values stored in difference form.
Contraction

• **ST-Trees** [ST83, ST85]:
  - first data structure to handle paths within trees efficiently.
  - It is clearly path-oriented:
    - relevant paths explicitly **exposed** and dealt with.

• Other approaches are based on **contractions**:
  - Original tree is progressively contracted until a structure representing only the relevant path (or tree) is left.
• Assume we are interested in the path from $a$ to $b$:

- Using only local information, how can we get closer to the solution?
Consider any vertex $v$ with degree 2 in the tree.

**Possibilities if $v$ is neither $a$ nor $b$:**

- $a$ and $b$ on same “side”: $v$ is **not** in $a\cdots b$.
- If $a$ and $b$ on different sides: $v$ **belongs** to path $a\cdots b$. 
Consider any vertex $v$ with degree 2 in the tree.

Possibilities if $v$ is neither $a$ nor $b$:

- $a$ and $b$ on same "side": $v$ is not in $a\cdots b$.
- If $a$ and $b$ on different sides: $v$ belongs to path $a\cdots b$.

We can replace $(u,v)$ and $(v,w)$ with a new edge $(u,w)$:

- This is a compress operation.
• Consider any vertex \( v \) with degree 1 in the tree:

- If \( v \) is neither \( a \) nor \( b \), it is clearly not in \( a \cdots b \).
Consider any vertex $v$ with degree 1 in the tree:

- If $v$ is neither $a$ nor $b$, it is clearly not in $a \cdots b$.
- We can simply eliminate $(v, w)$, reducing the problem size.
  - This is a rake operation.
Contractions

• A contraction-based algorithm:
  ▪ Work in rounds;
  ▪ In each round, perform some \textit{rakes} and/or \textit{compresses}:
    • this will create a new, smaller tree;
    • moves within a round are usually "\textit{independent}".
  ▪ Eventually, we will be down to a single element (vertex/edge) that represents a path (or the tree).
Path Queries

- Computing the minimum cost from $a$ to $b$: 

![Graph showing a and b connected through various nodes and edges with different weights.](image)
Path Queries

- Computing the minimum cost from $a$ to $b$: 

![Diagram showing a network with nodes and edges labeled with numbers. The nodes $a$ and $b$ are highlighted.]
Path Queries

- Computing the minimum cost from $a$ to $b$: 

![Diagram showing a weighted tree with nodes and edges labeled with numbers. The nodes are connected by lines, and the path from $a$ to $b$ is highlighted.]
Computing the minimum cost from $a$ to $b$:
Path Queries

- Computing the minimum cost from $a$ to $b$: 

![Diagram of a graph with points labeled a and b and edges labeled with numbers 2, 5, 6, 4, 3. The path from a to b is highlighted.]
Path Queries

- Computing the minimum cost from \( a \) to \( b \):
Path Queries

- Computing the minimum cost from $a$ to $b$: 

![Diagram showing a tree with nodes a and b and edges labeled 2 and 4]
Path Queries

- Computing the minimum cost from $a$ to $b$:
Dynamic Trees

Path Queries

- Computing the minimum cost from $a$ to $b$: 

![Diagram showing a minimum cost path from a to b]
Suppose a definition of independence guarantees that a fraction $1/k$ of all possible rakes and compresses will be executed in a round.

- All degree-1 vertices are rake candidates.
- All degree-2 vertices are compress candidates.

Fact: at least half the vertices in any tree have degree 1 or 2.

Result: a fraction $1/2k$ of all vertices will be removed.

Total number of rounds is $\lceil \log_{(2k)/(2k-1)} n \rceil = O(\log n)$. 
Contractions

- **rake** and **compress** proposed by Miller and Reif [1985].
  - Original context: parallel algorithms.
  - Perform several operations on trees in $O(\log n)$ time.
The Update Problem

• Coming up with a definition of independence that results in a contraction with $O(\log n)$ levels.
  ▪ But that is not the problem we need to solve.

• Essentially, we want to repair an existing contraction after a tree operation (link/cut).

• So we are interested in the update problem:
  ▪ Given a contraction $C$ of a forest $F$, find another contraction $C'$ of a forest $F'$ that differs from $F$ in one single edge (inserted or deleted).
  ▪ Fast: $O(\log n)$ time.
Dynamic Trees

Our Problem

• Several data structures deal with this problem.
  ▪ [Frederickson, 85 and 97]: Topology Trees;
  ▪ [Alstrup et al., 97 and 03]: Top Trees;
  ▪ [Acar et al. 03]: RC-Trees.
Top Trees

- Proposed by Alstrup et al. [1997,2003]
- Handle unrooted (free) trees with arbitrary degrees.
- Key ideas:
  - Associate information with the edges directly.
  - Pair edges up:
    - compress: combines two edges linked by a degree-two vertex;
    - rake: combines leaf with an edge with which it shares an endpoint.
    - All pairs (clusters) must be are disjoint.
  - expose: determines which two vertices are relevant to the query (they will not be raked or compressed).
Top Trees

- Consider some free tree.

(level zero: original tree)
Top Trees

- All degree-1 and degree-2 vertices are candidates for a move (rake or compress).

(level zero: original tree)
Top Trees

- When two edges are matched, they create new **clusters**, which are edge-disjoint.

(level zero: original tree)
Top Trees

- Clusters are new edges in the level above:
  - New \textit{rakes} and \textit{compresses} can be performed as before.
The top tree itself represents the hierarchy of clusters:

- original edge: leaf of the top tree (level zero).
- two edges/clusters are grouped by rake or compress:
  - Resulting cluster is their parent in the level above.
  - edge/cluster unmatched: parent will have only one child.

What about values?
Top Trees

- Alstrup et al. see top tree as an API.
- The top tree **engine** handles structural operations:
  - User has limited access to it.
- Engine calls **user-defined functions** to handle values properly:
  - `join(A, B, C)`: called when $A$ and $B$ are paired (by rake or compress) to create cluster $C$.
  - `split(A, B, C)`: called when a rake or compress is undone (and $C$ is split into $A$ and $B$).
  - `create(C, e)`: called when base cluster $C$ is created to represent edge $e$.
  - `destroy(C)`: called when base cluster $C$ is deleted.
Top Trees

- Example (path operations: \texttt{findmin/addcost})
  - Associate two values with each cluster:
    - \texttt{mincost}(C): minimum cost in the path represented by $C$.
    - \texttt{extra}(C): cost that must be added to all subpaths of $C$.
  - \texttt{create}(C, e): (called when base cluster $C$ is created)
    - \texttt{mincost}(C) = \text{cost of edge } e.
    - \texttt{extra}(C) = 0
  - \texttt{destroy}(C): (called when base cluster $C$ is deleted).
    - Do nothing.
Top Trees

- Example (path operations: `findmin/addvalue`)
  - `join(A,B,C)`: *(called when A and B are combined into C)*
    - **compress**: `mincost(C) = min\{mincost(A), mincost(B)\}`
    - **rake**: `mincost(C) = mincost(B)` (assume A is the leaf)
    - Both cases: `extra(C) = 0`
  - `split(A,B,C)`: *(called when C is split into A and B)*
    - **compress**: for each child `X \in \{A,B\}`:
      - `mincost(X) = mincost(X) + extra(C)`
      - `extra(X) = extra(X) + extra(C)`
    - **rake**: same as above, but only for the edge/cluster that was not raked.
Example (path operations: `findmin/addvalue`)

- **To find the minimum cost in path** $a \cdots b$:
  - $R = \text{expose}(a, b)$;
  - return $\text{mincost}(R)$.

- **To add a cost** $x$ to all edges in path $a \cdots b$:
  - $R = \text{expose}(a, b)$;
  - $\text{mincost}(R) = \text{mincost}(R) + x$;
  - $\text{extra}(R) = \text{extra}(R) + x$. 
Top Trees

- Can handle operations such as:
  - tree costs (just a different way of handling \textit{rakes});
  - path lengths;
  - tree diameters.

- Can handle non-local information using the \textit{select} operation:
  - allows user to perform binary search on top tree.
  - an example: tree \textit{center}.

- Top trees are implemented on top of \textit{topology trees}, which they generalize.
Topoqy Trees

• Proposed by Frederickson [1985, 1997].
• Work on rooted trees of bounded degree.
  ▪ Assume each vertex has at most two children.
    • Values (and clusters) are associated with vertices.
  ▪ Perform a maximal set of independent moves in each round.
  ▪ Handle updates in $O(\log n)$ worst-case time.
RC-Trees

• Proposed by Acar et al. [2003].
• Can be seen as a variant of topology trees.
  ▪ Information stored on vertices.
  ▪ Trees of bounded degree.
• Main differences:
  ▪ Not necessarily rooted.
  ▪ Alternate rake and compress rounds.
  ▪ Not maximal in compress rounds (randomization).
  ▪ Updates in $O(\log n)$ expected time.
Contractions

- Topology, Top, and Trace trees:
  - contraction-based.

- ST-Trees: path-based.
  - But there is a (rough) mapping:
    - dashed $\leftrightarrow$ rake
      - “this is a path that goes nowhere”
    - solid $\leftrightarrow$ compress
      - “both part of a single path”
  - ST-Trees can be used to implement topology trees [AHdLT03].
Chronology

- **ST-Trees:**
  - Sleator and Tarjan (1983): with balanced and biased search trees;

- **Topology Trees:**

- **ET-trees:**
  - Hensinger and King (1995);
  - Tarjan (1997).

- **Top Trees:**
  - Alstrup, de Lichtenberg, and Thorup (1997);

- **RC-Trees:**